Heuristics for Robot System Design using a Concurrent Hierarchical Control Model

John L. Michaloski, Ronald Lumia, and Thomas E. Wheatley
National Institute of Standards and Technology
September 1988

Abstract

The problem formulation of a robot control system is dominated by the time-critical aspect. To achieve success in this time-critical domain, multiple processor implementations are necessary. Multiple processor implementations of a robot control system implies that system designers must use a different set of design heuristics, strategies, and rules than those from a monoprocessing environment. This paper discusses traditional hierarchical structure as a real-time design methodology and proposes a concurrent extension of the model to include virtual control loops. Using task decomposition, this concurrent hierarchical design methodology attempts to solve the partitioning of system components across multiple processors.

The recent advent of Computer Aided Software Engineering (CASE) Tools offer the possibility to specify both the real-time and concurrent aspects of a system's functionality. However, these CASE only offer the semantics in which to model real-time behavior. These CASE tools do not offer the strategies, heuristics and rules to assist in the development of design model. As a part of the concurrent real-time control methodology, rules will be presented that guide the designer through different areas of design, including real-time constraints, multicomputer implications, interprocessor communication and synchronization. Finally, the current work at the National Institute of Standards and Technology will be presented to show how some of the design rules have affected the design and implementation process.

1.0 Introduction

Within the design process there exists the need to establish and model the relationship among components of a system with a formalized methodology. The hierarchical structuring of a system design is a common approach for showing the relationship among entities in a system. One method for constructing a structured hierarchy of a system is task

---

decomposition. T*ask decomposition* can be defined as the process of recursively breaking down a task into smaller, more manageable tasks, until some atomic level of activity is reached. Defining a hierarchical structure for a robot control system starts with a set of high level tasks and through a series of task decompositions reduces these tasks to a set of motion primitives.

As an example of task decomposition, assume a position-controlled manipulator with end-effector executes the command GET WRENCH, where the wrench is located at the logical position A and the desired logical goal position is B. Within this sample task, the command GET WRENCH is decomposed into a series of lower level commands GOTO, GRASP and RELEASE. The lower level command GOTO is responsible for mapping the logical to physical representations of the points A and B and producing a series of trajectories for the manipulator. The commands GRASP and RELEASE are responsible for controlling both the manipulator and the end-effector, and providing a logical to physical mapping for the object WRENCH. Using the notation of DeMarco [DEM79], this simple task decomposition would be pictorially represented as follows:

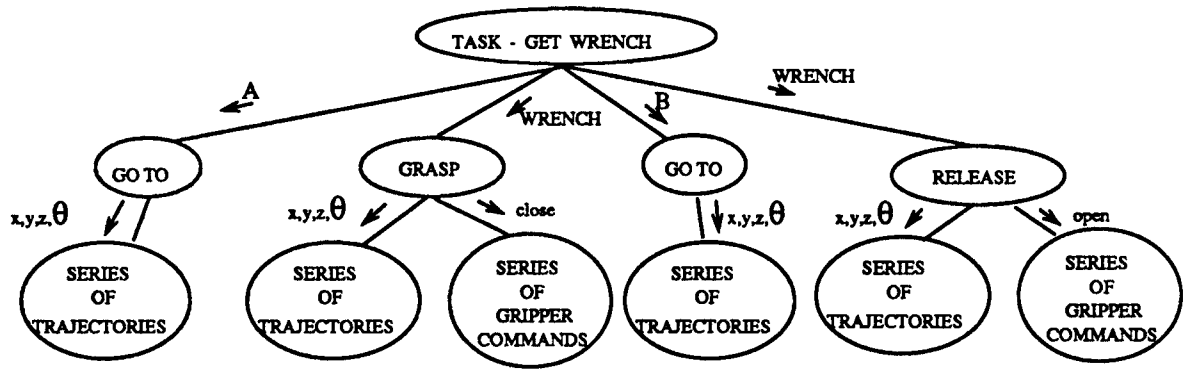

Figure 1. Task Decomposition into Structured Hierarchy

A standard set of generic levels of decomposition of a robot control system has been proposed at National Institute of Standards and Technology as a design methodology and is more fully explained in [ALB87]. Within this task specific example, the levels shown are the TASK level which corresponds to user command GET WRENCH; the commands GOTO, GRASP, and RELEASE map into the *elemental move* level; and the set of commands to move the robot and end-effector (i.e. gripper) map into the *primitive* level. Not pictured are the mapping of the series of trajectories into a *servo* level commands.

Performing task decomposition is a function of both the state of the world, and the number of operations that must be performed. This can get quite complex. Finite computing resources imposes the following fundamental robot system design and implementation rule: *responding to an event too late nullifies the control no matter how intelligent the subsequent action.* Because a robot control system operates in a time critical domain, a traditional hierarchical decomposition is not sufficient to define a control system. A structured hierarchy

only shows the partitioning of a system into modules, how the modules are arranged and the interfaces. It ignores the timing constraints and decision structure of the various components of the system.

This explicit timing constraint implies the need for a better methodology than a statically-linked hierarchical model of the entire control system. The National Institute of Standards and Technology has been using a concurrent modification of the hierarchical methodology to model real-time control for over 10 years[ALB81, BAR82, FIT85, FITZ85]. The methodology partitions levels of the hierarchy into individual control loops with fixed cycle rates and feedback. At each level in the basic hierarchical breakdown an interface exists where the adjoining levels exchange inputs and outputs as commands and status much like a feedback control loop. The fixed cycle rate implies a periodic sampling of commands and status which insures that global control flow is goal-directed. Unlike a purely sequential hierarchical decomposition, each lower level is not completely dependent on its neighboring upper level in this concurrent design. Although levels may share some data that models the world, each level can be considered to run independently of each other, responding to a command, and supplying a status much like a plant in a normal feedback control system.

Modifying the previous task decomposition the following diagram now statically illustrates the concept of concurrent hierarchy. Again using the notation of DeMarco [DEM79], the task decomposition is pictorially represented as follows:

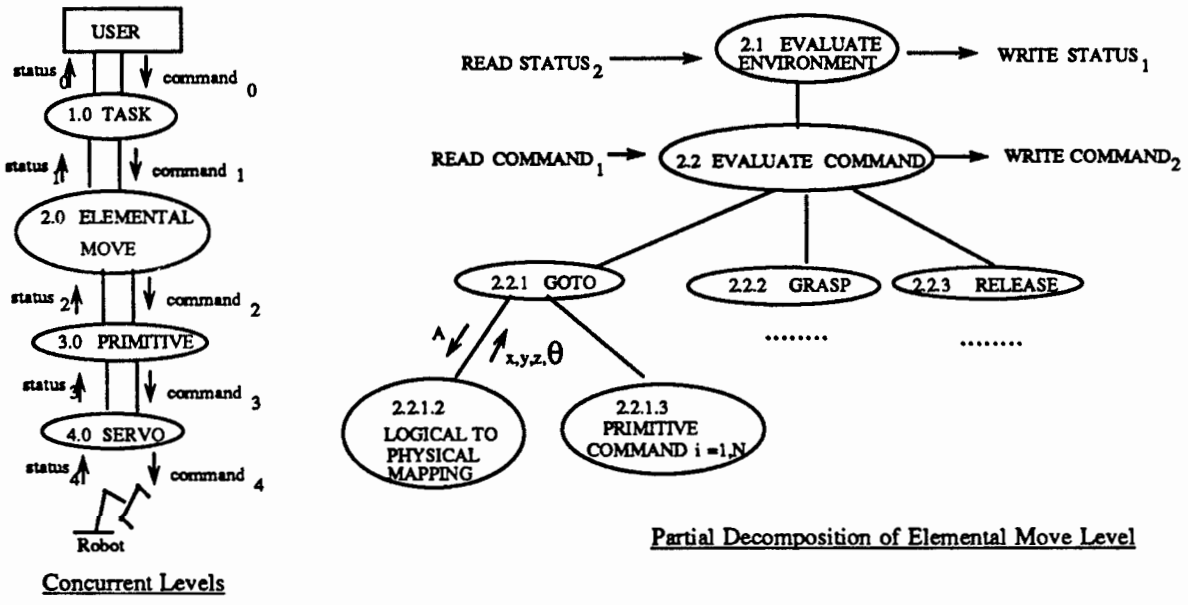

Figure 2. Concurrent Task Decomposition

This time the task decomposition partitions the robot control system into four concurrent levels. The elemental move level is partially expanded to illustrate the mechanism in which the levels operate. For each cycle, a level reads a command and status, evaluates the command and environment, and then determines a course of action which is output as a command and status to adjoining levels.

When composed as a system, concurrent hierarchical control can be defined as layers of *virtual control loops*. When executing, each virtual control layer in the hierarchy can be considered as part of a long chain defining the hierarchical state, yet each level's action is based on its own control flow. Much as a computer executes an instruction within a given duty cycle, the virtual control loops correspond to layers of software modeled analogously to a physical machine. The virtual control loop software exhibits cyclic feedback behavior that samples inputs including command and status and guarantees some output within a given time.

The underlying assumption of a fixed-cycle response time within the system restricts the overall processing time for sophisticated control. There is only so much processing a level can perform within each fixed cycle. This leads to the motivation to further divide task decomposition concurrently into planners and executors. The *planner* is responsible for generating a plan consisting of a series of actions. The "best" plan is selected from a candidate list of alternative plans that achieve the commanded goal, given the current state of the environment. An *executor* is responsible for stepping through a generated plan. The executor matches the current state of the machine against a set of preconditions as in a production system, which triggers the corresponding action. Now, each level can maintain real-time control, yet concurrently evaluate alternative future actions.

The planner and executor together compose the task decomposition module that determines the control behavior. But there are more functions to feedback control than just planning and executing. Supplying and interpreting external sensor information is another important function in feedback control. Sensor processing operates concurrently with the control function to interface to the real world. Sensor processing consists of data acquisition (reading sensors) and data qualification (translating raw sensor data into engineering units and filtering bad data).

The disparity of purpose between control and the sensing leads to a function to mediate the information exchanged between the control and sensing functions. The mediating function, defined as the world model [ALB81], state knowledge or global data base in a production system [WIN84, NIL80], allows the decision process to become more abstract in its nature of queries about the world, and less dependent on the physical nature of the actual sensors being used. As a supervisor of the systems data base, the world model provides the system's best estimate and evaluation of the history, current state, and possible future states of the world. The world model provides accurate and current information for reflexive behavior (i.e. what is), the best estimate of the world in deciding the future plans (i.e. answer what if), integrates the information from the sensory processing module. This leads to a vertical partitioning of the virtual control loops into control, sensing and world modeling components. Figure 3 illustrates the generic partitioning of one level within the hierarchy.

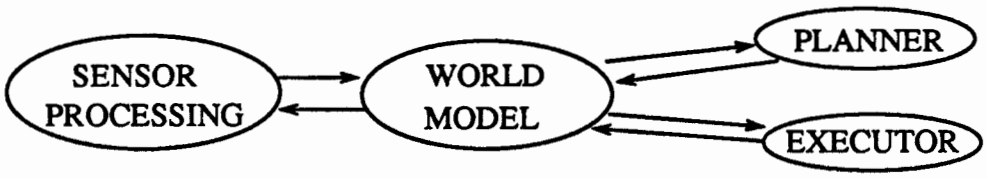

Figure 3. Control Level Partitioning

Further refining and partitioning of a control level is possible, but is beyond the scope of this paper. Albus more thoroughly covers the decomposition and partitioning of a hierarchical control system [ALB87]. This paper will use a coarser grain of partitioning to study basic requirements applicable to all concurrent real-time system design.

2.0 Characteristics of a Real-Time System

A traditional real-time software development environment supports the development and testing of algorithms for logical correctness on a "slower" *host* machine. Subsequently, these logically correct algorithm are downloaded to a "faster" *target* system and tested under real conditions. Target systems operate in the real-time domain which implies a premium on efficiency. Real-time systems for monoprocessing environments have well-established performance requirements. In general, real-time systems are characterized by performance that includes high data communication bandwidth, maximized throughput, and fast execution. To realize these requirements, real-time target systems make use of the following features:

- fast context switching time between tasks
- small interrupt latency
- feature priority scheduling
- allow high-priority tasks to instantly pre-empt lower priority tasks
- grant low-priority tasks non-preemptable status
- methods for synchronization of tasks

However, this does not define a real-time system and leads to the following design rule: *reliable and deterministic performance within the time-critical domain are the necessary requirements that define a real-time control system.*

Implicit in this design rule are other unique real-time processing corollaries ignored in a general purpose electronic data processing design methodology. The following design corollaries are based on the need for deterministic processing time estimates and include:

> Rule 1. *Fairness is not an issue in real-time control.* In a general purpose operating system, every process is given an equal opportunity at the CPU. At the application level in a real-time control system either the processes are permanently dedicated to the hardware or are guaranteed to be resident during critical sections (i.e. non-preemptable). This is not to imply that some processes should be

processor starved. Rather, scheduling must be established statically beforehand to insure real-time execution.

Rule 2. *Tasking must be deterministic and user-controllable, thus time slicing is less important.* In general, round-robin scheduling algorithms are nondeterministic and therefore difficult to characterize execution. Difficulty in characterizing execution implies difficulty in verification. Exact time quantum round-robin scheduling can overcome the non-determinism but the overhead of continual context switching overrides the use of the round-robin except in very low speed applications where response time is not critical. Instead, a priority based system where processes execute in known sequence, surrender the processor when finished, or are blocked awaiting an event, is preferred.

Rule 3. Virtual memory is not as important. An operating system that supports memory management of a virtual address space is not required since the operation of swapping memory in and out from disk is slow, and the price of memory is cheap.

Rule 4. Dynamic creation of processes is of limited usefulness [SCH87]. The overhead required to replace unreliable or crashed processes by new processes is too large.

2.1 Design Considerations Due to Constraints Imposed by the Real-Time Domain

Design methodologies for a sequential electronic data processing environment propose the use of small module size, modular independence, black box definition of modules that hides implementation from purpose, and isolation of logical detail from the physical implementation. The time critical domain of real-time control software is well-served by these design goals but adds additional constraints that demand a more rigorous accountability in the design. The designer must be much more careful and exacting in the detail of the specification. In order to establish real-time reliability, a stronger threshold of assurance than just logical program correctness is required. Much as structured programming helped improve the quality of sequential software produced, a few simplifications to a real-time controls system help improve its quality. This section details some simplifying assumptions about control system which reduce the software complexity, the impact of timing and required efficiency necessary to meet program objectives.

The design concept to use layers of abstraction in which a program is hierarchically divided into a set of subprograms is a commonly accepted programming practice and much of structured programming is based on this hierarchical structuring concept. In a purely sequential machine, most structuring is done with information hiding where as much of the code is treated as a black box as is possible. The user knows how the box functions, but does not need to know how it was implemented. Unfortunately, the nature of the abstractions that may be conveniently achieved in a purely sequential machine do not work with a real-time control system.

In real-time systems, the user may not need to know how it was implemented, but must know what the black box will do, how long it will take, and what it will do with

unanticipated problems. Again, this is the deterministic criterion faced in real-time design. One simplifying design rule in order to maintain a functioning system requires that *no modules should contain code that waits on an anticipated event.* This type of infinite waiting can hang any system, be it sequential or concurrent. For example, how many times has a program failed because a while loops condition doesn't terminate. In a real-time control system waiting on a dead sensor for a reading is the analogous mistake. Instead of waiting on a condition, the control code should sample the environment and act accordingly. If the event has not occurred, measure how long the duration has been waiting for this event, and if it exceeds a reasonable time period flag it as an error. Then a higher level can evaluate whether the system can still run without the use of this module. If one safety sensor fails, that's acceptable, if several safety sensors fail, and their backups fail, immediate system shutdown may be the only recourse.

Sensor sampling has another profound effect on the specification. How the control system responds to external events can be classified as either demand or periodic functionality [BRI79]. A demand function is triggered by the occurrence of some event every time it is performed. For example, a trip switch may only be activated intermittently, but could be triggered successively at indeterminate intervals. A periodic function is performed repeatedly without being requested each time. For example, a vision system may supply a new image at fixed intervals, regardless of whether the system requested the new image.

One design simplification that provides for a more rigorous specification of a control system is to *sample all sensor or real-world inputs at a periodic rate, and removing demand functions and thus unpredictable behavior.* Restructuring demand functions as periodic functions can be done deterministically with the use of latches and polling, and repeated sampling each control cycle. Although polling is wasteful, the time required for a control cycle to sample all the sensors can be realistically bounded by a fixed time period. This also has the effect of removing the racing conditions that typically result within a control program as a result of heavy workload when sensors burden the system. Anticipating this workload by sampling all sensors eliminates the racing condition and prevents unforeseen overload and/or system crash.

Assuming the control system has a limited number of sensors, that can be sampled within a relatively short time frame, this constraint produces a uniform arrival distribution of sensor updates. However, even systems with a large number of sensors should use this approach. Systems that do not use this simplification and have too many sensors to sample within a reasonable control cycle must assume a Poisson arrival distribution and determine a lower bound on the interval between consecutive sensor signals in order to establish functional correctness. Given a Poisson distribution for interrupt arrivals, one cannot guarantee every sensor reading. With a periodic sampling of sensor methodology, one can calculate the minimum repetition rates across all sensors to determine the minimum operational control cycle necessary. Should this control cycle not have enough processor time left over after sensor sampling to do processing, then the sensor sampling may have to be divided across processors and the use of a sensor fusion module would have to be used.

3.0 The Effect of Concurrency and Parallelism on Design

In order to satisfy the real-time timing constraint of a robot control system, the standard hierarchical structure model has been modified to include a formalized methodology for concurrent design. A major advantage of this concurrent model of the robot control is that it is easily ported to a parallel implementation. So far, strictly mono-processor, real-time design issues have been developed. Extending the control system to include parallel or multi-computer solutions has a major impact on the design methodology. The major rule governing any effective multiprocessor system design is *to exploit the benefits of parallelism while minimizing the impact of parallelism on the software algorithms.*

A robot controller can be characterized as a speedup-oriented multiprocessing application since the controller is partitioned into a set of concurrent, cooperating processes. Because there are a large variety of multicomputers available that offer dramatic parallel performance, the methodology presented will focus on those parallel architectures that best fit the system requirements of an intelligent robot controller in terms of price versus performance. A multiprocessor system sharing a common or enhanced bus is an example of an architecture that offers a mix of capabilities that can accommodate the necessary architecture diversity. These real-time multicomputer architectures require additional features that support parallel operations including:

- coherence of cache memory
- interprocessor synchronization
- interprocessor communication

Coherence of cache memory is beyond the scope of this paper and can be viewed more as a required hardware design feature as opposed to a software design issue. Interprocessor synchronization and interprocessor communication can be viewed as part of the same problem; how to design concurrent software that is reliable and correct. The pair can be viewed similarly, except one is exchanging control as opposed to data information. Typically, synchronization is tied more closely to the hardware of the machine and the communication models are built using the synchronization primitives. This following sections will limit the design discussion to the software aspects of communication, plus the effect synchronization on communication.

3.1 The Effect of Interprocessor Communication on Design

The basic design rule for interprocessor communication *is that it must be efficient enough to justify the additional overhead of communicating between processors, otherwise the extra processors are extraneous.* To evaluate the effectiveness of interprocessor communication, the performance measures of latency and throughput are used. To characterize a robot control system as real-time implies a guaranteed maximum latency for interprocessor communication. *Latency* is defined as the elapsed time before a message is acknowledged. *Throughput* is defined as the number of bytes one process can send to another process in one second, especially important for systems that share large amounts of data. When designing, the distinction between guaranteeing arrival of small amounts of data every 20 milliseconds must be contrasted to transferring large amounts of data efficiently across the transport link. Thus, 20 bytes of information that has to be shipped every 20 milliseconds cannot be handled with a communication protocol that is efficient for moving

large amounts of data, but may require 100 milliseconds setting up the physical link handshakes and transport software headers.

This leads to the following rule that *guaranteed performance i.e. latency, not efficiency i.e. throughput, is the ultimate measure of real-time communication.*

The concepts of flexibility, data integrity and extensibility are equally as important in evaluating a communication scheme, but are less tangible. Data integrity and flexibility can best be explained with an example. On sequential machines, interprocess communication is performed by calling or invoking a subroutine and passing the appropriate parameters on the stack by value or by sharing global variables. How the parameters are exchanged controls the method of communication. Passing parameters "by value" places a copy of the parameter onto the stack and can be considered message passing since each processes' internal variables are independent of the other processes. Such a method of communication can be considered message passing. Passing parameters "by reference" pushes a pointer or address onto the stack so that data is shared between the processes. Such a method of communication can be considered a shared memory technique. In a monoprocessing architecture, passing parameters "by reference" saves excessive copying. However, in parallel processing, such copying is more flexible since a user cannot be assured that the two processes both have access to the pointers or addresses, and that the addresses are valid. For example, in a dual ported memory scheme, on-board versus off-board addresses differ.

This subtlety raises a real-time control design dichotomy, although complete copying offers better flexibility and data integrity; it is slower and therefore infeasible for many design constraints. Within parallel architectures, this communication dichotomy is modeled as a tightly or loosely coupled approach. Hybrid model approaches combining aspects of both form of communication are also quite common. Figure 4 illustrates the coupling of memory and processors in each communication scheme.

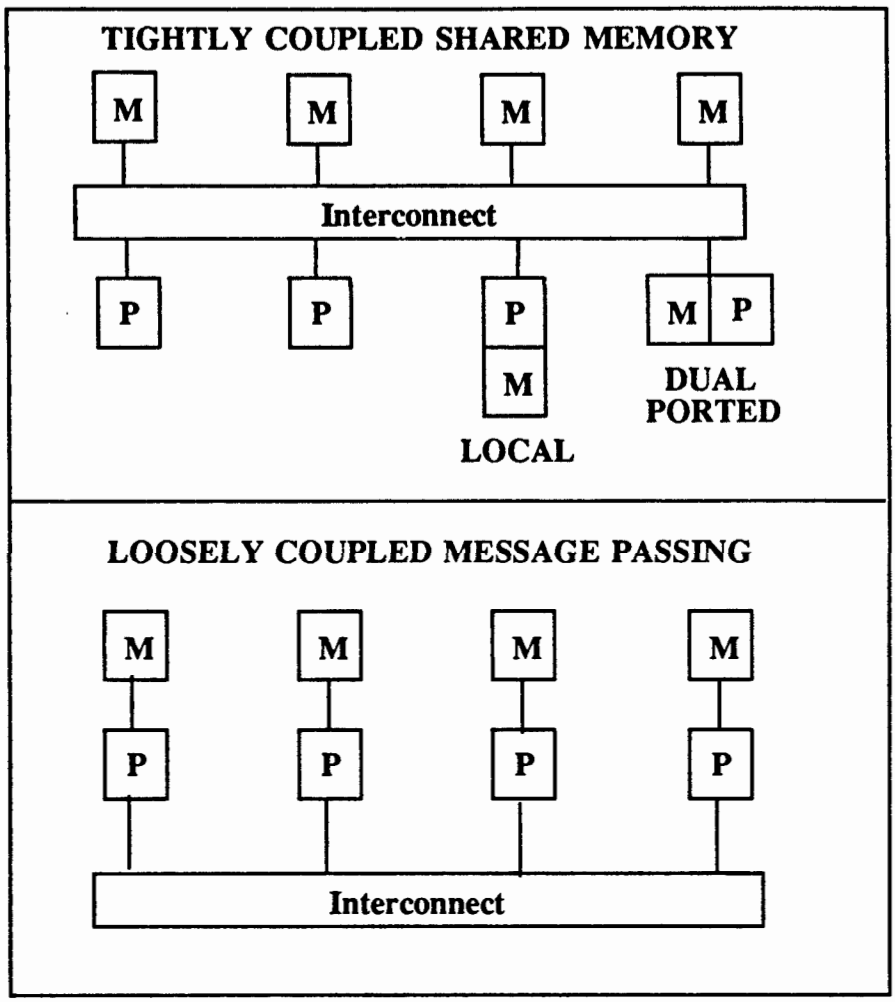

Figure 4. Tight versus Loose Memory Coupling

Loosely coupled systems have private memories where processors generally cannot read each others memory and must have an explicit *message-passing* communication channel. Conceptually, message passing can be considered passing information by value. *Message passing* systems have memory attached privately to each processor (at least conceptually), so that processors communicate only through explicit transmissions of whole messages [LYO87]. Message passing is more generic in that messages can be passed not only across backplanes but across machines.

Tightly coupled systems have a public memory architecture, where processors communicate through *shared memory*. Conceptually, shared memory communication can be viewed similarly to passing parameters by reference. Shared memory communication provides equal access to all processors to the shared memory. Shared-memory offers many advantages, including ease of sharing data, rapid communication and high performance. The advantages that make shared-memory architectures attractive result from the tight coupling along the critical path between processors and memory. This leads to a notable increase in

performance. Shared memory offers such a highly efficient and straightforward method for communicating among parallel processes that is commonly used for parts of real-time systems [PAU86, KOR86, KAZ87]. Moving large amounts of data or shifting processes to and from private processor memories can be cumbersome and slow. A shared memory scheme is more appropriate here since attempting to use a message passing technique as a server to mediate shared data can be much slower than direct access [KAR87] (factor of thirty). The disadvantages that arise from a shared memory communication scheme include: 1) guaranteeing mutual exclusion of shared resources across processors, 2) as the number of processors grow, contention in the form of arbitration of accesses to the common memory degrades system performance, 3) maintaining a consistent view of absolute, physical addressing across processors, and 4) shared memory models are not easily portable.

This leads to the following communication design rules: *restrict shared memory for very high-speed specialized design purposes such as moving very large amounts and/or time-critical data between processes.*

3.2 The Effect of Data Abstraction on Communication and Design

Many times the distinction between a shared memory implementation and a message passing system is not explicit but is based on the level of abstraction of the data and implementation. From the implementation standpoint, suppose two processes share a variable using a semaphore, mutual exclusion of a variable is provided through semaphore signals before each read and write. This shared memory exchange can be considered message passing. The subtle distinction arises in that shared memory schemes are application-specific and implemented with little operating system intervention while message passing is usually considered a part of the overall operating system.

The level of abstraction of the data shared between processes strongly effects the communication design. The most straightforward algorithm is double buffering using shared memory where only one reader and one writer grab complete control of the common memory until finished when restriction to common memory is removed. In this case, each communication partner must explicitly know the representation of the data. Exact knowledge of the physical data representation and manipulation is unwanted because it forces more of the physical details of the machine into the design. The client/server model overcomes this problem. This method fits into a more classical message passing type of interface in that it adds another layer of abstraction, and is therefore slower. The server offers a level of abstraction from the user (i.e. the client) who queries the server. The server hides the client from the physical implementation details and allows the level of discourse to be of a logical and abstract nature. The client/server offers the strongest rationale for using a message passing scheme (even if it simply a subroutine call). Within the control hierarchy, the world model acts a server to its clients, the planner, and the executor. Clients request service from the world model. If the world model is busy, this request is queued. When ready, a client is serviced by the world model. The world model translates all logical queries concerning the system into a corresponding physical representation and responds with an answer. Thus, the world model shields its clients from understanding its physical representation of the world.

Given these two communication interfaces, this general design rule applies :

> *A server is necessary when one process views communication information logically, while the other process is closely tied to the physical representation. Further, the communication of information that is based on a complex data representation imposes a large burden of programming responsibility on the client. A server is necessary in order to shield the client from complex information access.*

3.3 The Effect of Synchronization on Interprocessor Communication and Design

Synchronization is a basic building block that a multiprocessor system must contain. Synchronization serves the dual purpose of enforcing the correct sequencing of processes and ensuring the mutually exclusive access to certain shared, writable data. Synchronization is usually supported by some special-purpose hardware. These basic synchronization primitives are then used to build higher level synchronization and communication tools in software or microcode. Communication and synchronization are difficult to separate because communication primitives can be used to implement synchronization protocols, and vice versa. The amount of synchronization leads to the characterization of communication mechanisms as either synchronous (or blocking) or asynchronous (or non-blocking). Routinely, blocking implies surrendering the processor and waiting for some condition to occur before continuing execution. Whether to block provides the fundamental design distinction between a synchronous and rigorous communication exchange and an asynchronous and efficient communication exchange.

Synchronous communication protocol uses an explicit handshake for acknowledgment. Both the sender and the receiver synchronize, the message is sent, and the sending processes waits until it receives an acknowledgment. In an asynchronous communication protocol, the sending process does not have to wait or block for the receiving process to acknowledge receipt of the message. Since asynchronous messages can arrive at random times, queues are attached to save messages. Acknowledgment is optional, but can be installed as part of the mechanism. The message itself must contain instructions whether an acknowledge is necessary.

Synchronous message passing primitives combine process synchronization with information transfer. Two processes first synchronize, then one process transfers information to the other, and finally each continue their individual activities. This synchronization is called a rendezvous. The programming language ADA uses this style of communication. Another style of synchronized communication is the extended rendezvous or transaction concept from Concurrent C [GEH86]. In a simple rendezvous, the exchange of information is unidirectional- from the message sender to the receiver. An extended rendezvous allows bidirectional information transfer using only one rendezvous. Thus with each transaction both processes exchange information. Figure 5 graphically illustrates the differences between asynchronous and synchronous communication.

ADA is a registered trademark of the U.S. Government, Ada Joint Program Office.

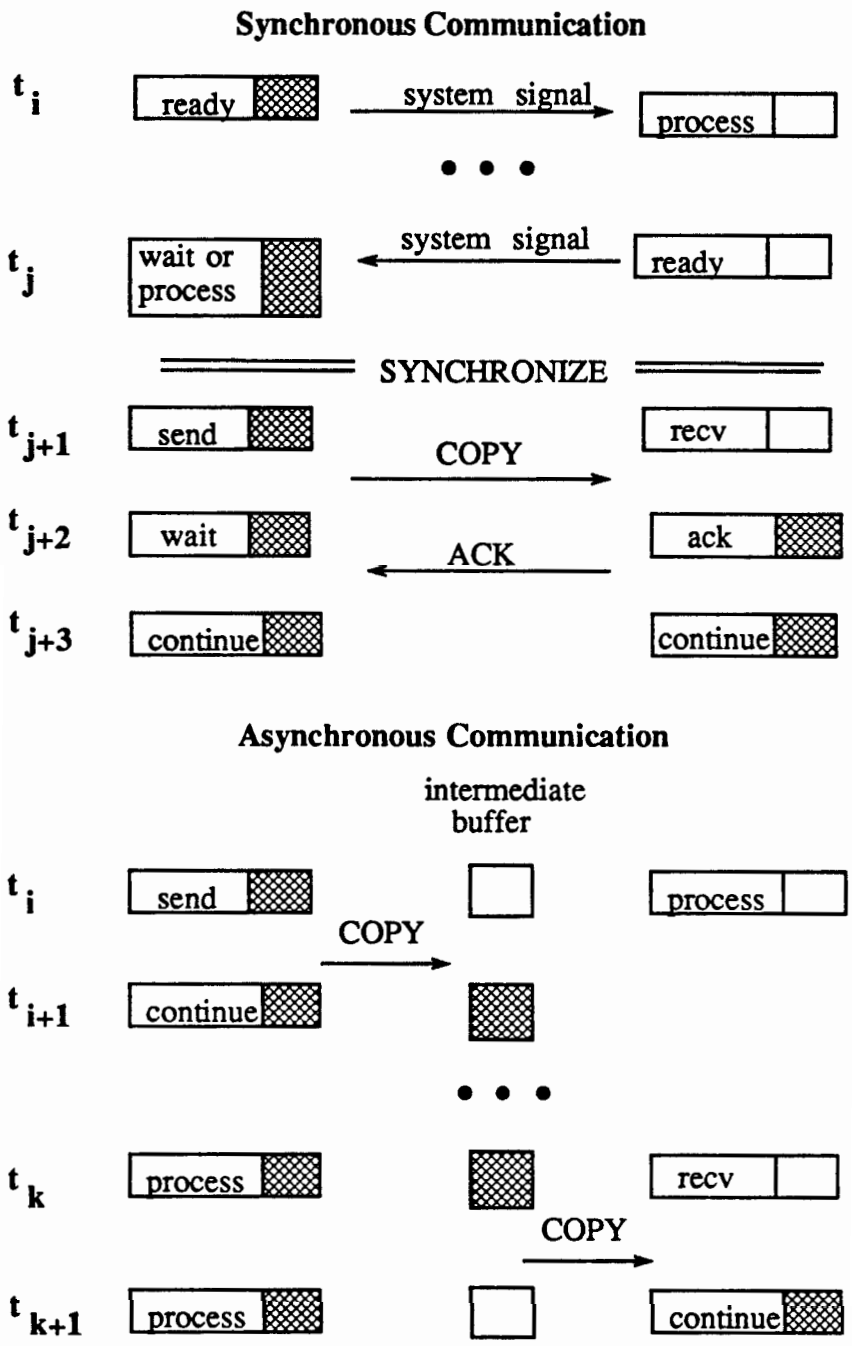

Figure 5. Synchronous vs. Asynchronous Communication

One advantage of synchronous communication is simplicity (no queues or queuing monitor) and thus low overhead. Another advantage is the savings in space and time that result because data can be directly copied from the sending processes' buffer to the receivers with no intermediate storage action required. Finally, the one-to-one correspondence of a synchronous transmission provides a stricter notion of accountability and determinism that

enhances software reliability. The major disadvantage of synchronous communication is the computational overhead spent synchronizing and acknowledging, i.e. waiting, for each communication. Another disadvantage is the lack of flexibility and subsequent extra software penalty for handling dynamic reconfigurations such as many-to-one communication channels. For example, in a dynamic environment, a synchronous communication exchange requires a priori knowledge by both parties of the other's existence in order to synchronize.

The advantages of asynchronous communication are speed and flexibility. The lack of synchronization and acknowledgment steps improves performance. The disadvantage is that error detection may be overlooked. Flexibility results in that the message may embody more of the communication mechanism, i.e. destination of a response of acknowledge. With information embedded in the message rather than the a priori synchronous connection, a new client can be dynamically added to a server by including a destination address within the message. The disadvantages of asynchronous communication is the lack of accountability for errors thus requiring programmer discipline to foresee and handle errors.

These two models are basically equivalent, in that the synchronization (i.e. control information exchange) before the communication can be considered a bi-directional asynchronous communication. Further, asynchronous communication is usually extended to include an acknowledgment step. In addition, synchronous communication can use a table lookup for dynamic reconfiguration. This leads to the following design rule: *from a software validation standpoint, synchronous communication is preferred, but the system may not tolerate the extra amount of overhead.* In general, for connections that are statically predefined one-to-one exchanges, synchronous communication provides the most reliable scheme. For connections mapped as many-to-one and dynamically reconfigurable, asynchronous communication provides the cleaner but less rigorous design.

3.4 The Effect of Communication Connectivity on Design

Within a message passing system (possibly implemented via shared memory), coordinating the location of the receiver of a message is an important design issue and will be termed connectivity of the exchange. Connectivity can be either temporary, known as datagrams, (i.e. for the life of the transmission), or permanent, known as virtual circuits (where the communication is ongoing until the channel is closed much like file manipulation activities) [POS80]. Interprocessor connectivity can be established dynamically through a system broadcast [FRI87] or by the more common, statically defined connections [NAR86]. A dynamic connection is established by the sending process broadcasting a system-wide request for the receiver location, who responds with its location, and then communication is direct.

Dynamic connections offer flexibility, but the overhead reduces performance. A hybrid alternative for permanent communication links proposes the use of dynamic communication connections at system start-up to establish the connectivity. Dynamic connection for temporary links would not be useful for a system with stringent real-time control demands. Static connections use tables of some kind to link sending and receiving processes. Static connections are fast, but require the logistical overhead of some centralized server to map the logical to physical locations.

The amount of connectivity is another issue. Given a message passing system, whether to decentralize message service and attach a message exchange with each process or use a centralized message server for a family of processes is another design consideration. Which message server approach depends on the system performance. If speed is a concern, numerous message exchanges would require unnecessary overhead. Each time a process must wait for a message, it blocks; this requires a context change between processes. With numerous message exchanges, CPU performance would lag burdened by the continual context swapping between processes. A centralized approach reduces the context switching overhead, yet requires an encoding system to describe the destination of the final message.

Decentralized connectivity is more flexible, but imposes a larger system administration problem. Each message exchange must have its own queue and may be more difficult to verify response performance. This approach would be more useful for high-level demon type processes with infrequent activity. Only running occasionally, such as in emergency situations, these demons are best treated as processes independent of many algorithms, and part of a set of general purpose tools commonly shared among all algorithms.

The basic design rule applicable to connectivity is to *centralize modules that depend on each other and decentralize modules that are independently coupled. In the case of a fault-tolerant module, processes that communicate across processors would require dynamic and decentralized connectivity.*

3.5 The Effect of Data Loss on Communication

Initially, general communication can be considered simply as a two part process - writing and reading. One process writes information, and another process reads this information. Interprocessor communication can be characterized as a read-write sequence with two additional facets, synchronization for the exchange of information, and destructive versus non-destructive read/write execution. For example, writing to queue is non-destructive (assuming enough storage). Writing to a variable is destructive. Removing an item (message) from a queue is a destructive operation. Reading a variable's value is a non-destructive operation. Figure 6. summarizes the combinations of non-destructive versus destructive execution during communication and the styles of communication that result.

	Destructive Write	**Non-Destructive Write**
Destructive Read	non-queued message passing	queued message-passing
Non-destructive Read	variables, shared memory	mail (assumes explicit delete mechanism)

Figure 6 . Styles of Communication

The execution may also have a time-out feature, so that the reader (or writer) does not wait indefinitely for a new information. The information exchange can be synchronous (both

actively participate in the exchange at the same time) or asynchronous (where the information exchange occurs non-deterministically). Communicating task interaction can be identified in real-time software as asynchronous with data loss, synchronous execution without data loss, and synchronous or asynchronous operation with possible loss of aged data [SCH85]. Program verification between communicating processes is complicated by data loss. Synchronous communication is the preferred communication technique since it encourages accountability between communicators. However, a stringent performance may dictate asynchronous communication and the subsequent possibility of data loss. This leads to the following discussion on various data loss communication issues.

a) How should the receiving process acknowledge a new message from the sending process? For example, with each level running independently in a virtual control loop, how often should the command/status handshakes acknowledging execution be required? Should there be a one to one correspondence between commands and status or the latest command or status be supplied as often as possible? Too much handshaking with a synchronous protocol (for example, upon each message exchange) slows system performance considerably. However, for a state machine implementation, one to one correspondence between a command and a status must be provided. In this case, a general purpose design rule within the concurrent hierarchy incorporates a *time stamp embedded within a command.* This time stamp is acknowledged by the lower level by returning the current input command with time stamp embedded within the returning status. This design produces a flexible, yet efficient, means to accomplish the verifying handshake.

b) How long should a process wait for a message? Should the receiving process continue operation using the latest message again? As a simplifying system assumption, the design rule within the concurrent hierarchy to use *one to one synchronous message exchanges, especially for interlevel communication, should be used to encourage a state transition machine that provides a deterministic execution trail.*

c) If both processes are running asynchronously and the sending process is generating more messages than the receiving process can consume, should messages be queued or overwritten?

Most message passing facilities assume the programmer desires queuing of messages. However, if levels were to run completely asynchronously within the hierarchy, messages could queue up. This leads to the design rule within a concurrent hierarchy to have *neighboring higher levels programmed to delay issuing a new command to the neighboring lower level until the previous message has been acknowledged in the returned status.* This method of communication is dictated on the premise that the executor is sampling commands and supplying status at known intervals at time throughout the system.

4.0 Analysis of System Software Requirements

The transition from a concurrent design to a multicomputer implementation includes the

need to map processes to processors. Should the granularity chosen not realize the desired speed, additional processors may be required, or revision of the algorithm on the host system for subsequent use on the target system may be required. Because of the iterative nature of defining the allocation of processors and scheduling of tasks, the design methodology must be based on the need to meet timing requirements. In the case of a robot control system, understanding the disparity of timing constraints among the different concurrent hierarchy levels is important in understanding the varying degrees of parallelism required.

At the very lowest levels of the concurrent hierarchy, the functional components must be efficient and highly streamlined such that parts of the control system may have to run without the luxury of any operating system assistance. A low level servo controller cannot afford the time to allow multi-tasking or other system overhead since updates are striving for millisecond updates, and microseconds are preciously few. Processing at this level requires stand-alone processing with little or no interaction with an operating system. A kernel which handles spurious interrupts, limited background I/O service and a basic monitor *for handling off-line board level troubleshooting is sufficient for streamlined system support.* However, stand-alone levels still require interprocessor communication to other levels in the hierarchy.

Higher levels are allowed longer processing intervals so that context switching using multi-tasking among different processes can be performed. Further, higher levels may require a high-speed file system for data logging and performance analysis that can be used as an evaluation tool or as a postmortem data recorder box (or "black box") after an unforeseen crash of the system.

Review of system requirements suggests the need for *a computing system with a rich set of software tools that supports a broad variety of processing needs ranging from modeling the real-time hierarchy to supporting the user interface.* This rich tool set should feature system software support equipped with interprocessor communication, synchronization, multitasking, preemptive and priority scheduling, resource (including memory and file) management. Figure 8 illustrates the set of programming tools necessary in a hierarchical real-time control system operating on a parallel pipelined system architecture. Note that the location of memory is not important because interprocessor communication (ipCOMM) is concerned with data flow, independent of the actual communication means; be it high-speed bus, generalized communication network or shared memory.

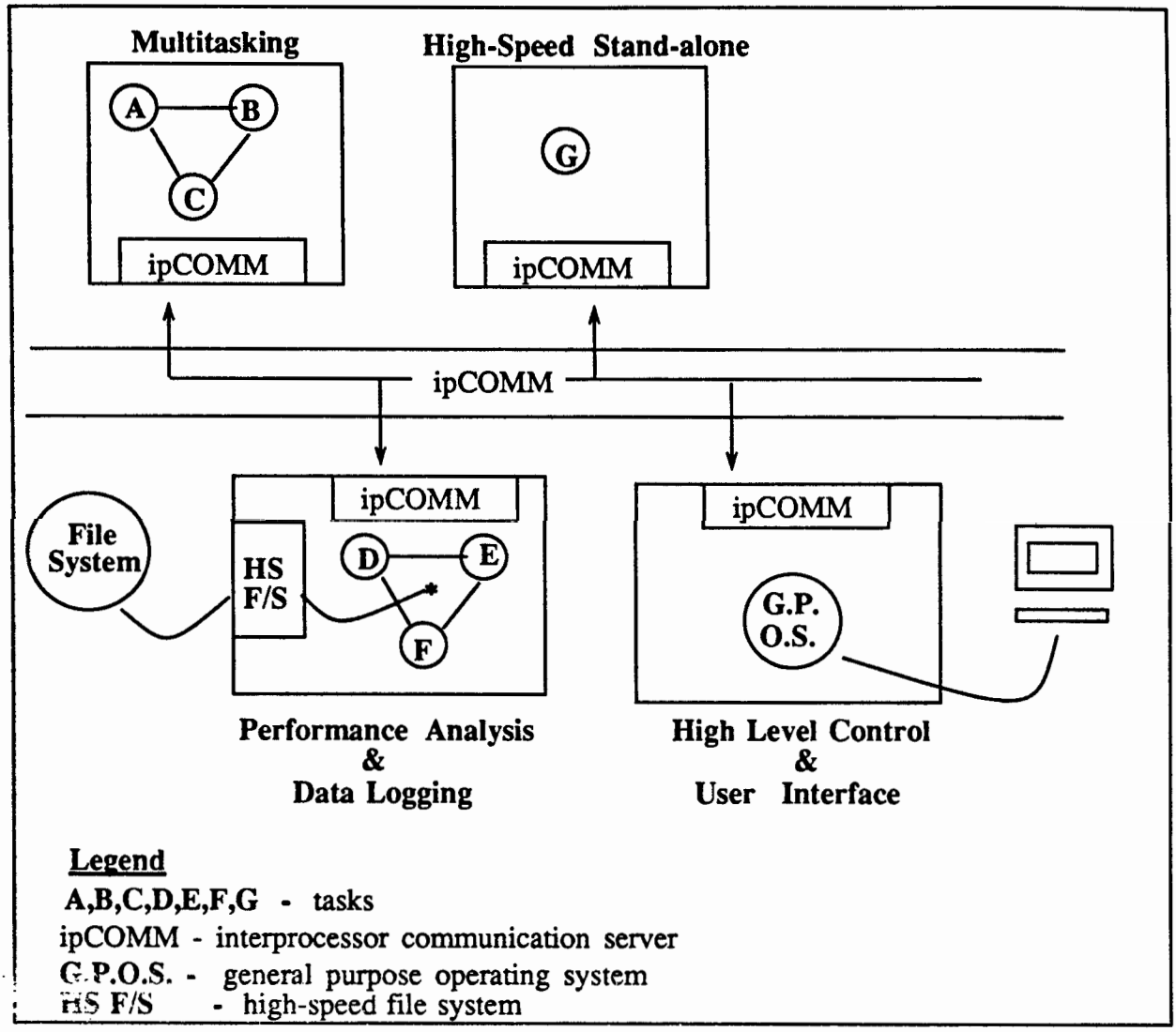

Figure 8. Programming Environment

This prototype architecture for hierarchical control exhibits differing styles of *concurrency including communicating parallel processors execution or interleaved execution* on a single processor. Because of the disparity of response times required at various levels in the hierarchy, different levels of granularity are required. At the lowest levels, planning may be impossible, and even execution may require multiple processors to achieve a solution. Moving higher in the hierarchy, timing constraints prevent the planner and executor from residing on the same processor. In this case, the two processors would run in parallel and asynchronously the planner would update plans. At higher levels, completion time for plans is less critical; so that multitasking on a single processor can supply enough processing power for both the planner and executor. Further, if the timing constraints are not stringent, multiple levels can be combined onto one processor, or the use of a local area network can be used to connect other computers as a part of the controller. Each of these configurations is based on the response time requirement of the level. Figure 9 provides a

guideline for the type of performance required of differing levels in the hierarchy.

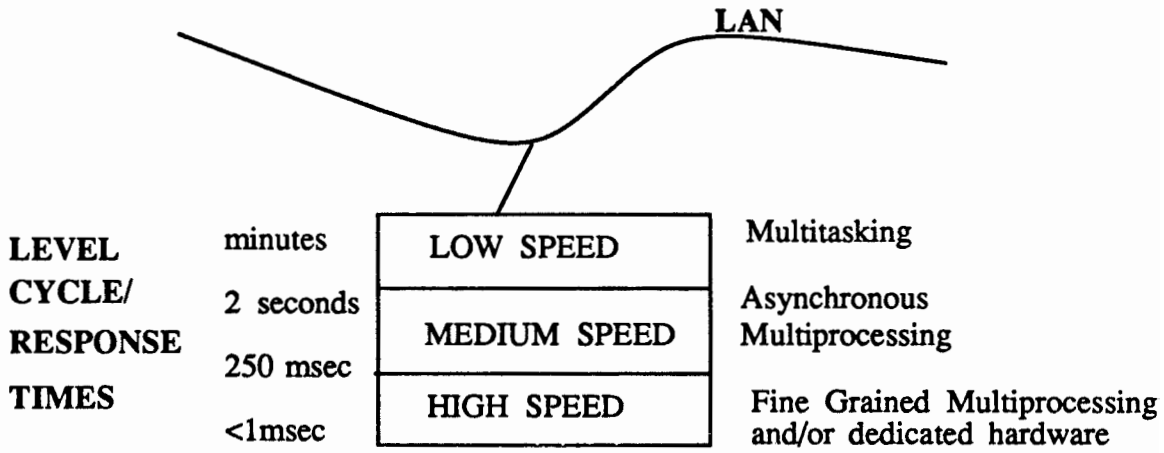

Figure 9. Computing Resource Utilization Based on Cycle Response Time

4.1 Survey of Existing Design Architectures

A survey of existing multiple processor robot control systems covers a wide range of implementations. The requirement to satisfy real-time constraints has lead to numerous implementations using tightly coupled architectures with several processor boards on a common bus [PAU86, NAR86, KOR86, KAZ87, GAU87, SCH87]. The target operating system of these implementations vary in degree of system complexity from the basic use interrupts and handshaking operations to handle interprocessor communication[PAU86], to a limited operating systems with special communication features [NAR86], to an extended real-time operating system [SCH85].

The aforementioned implementations share common system architectural characteristics with a concurrent hierarchical control system. Although few of these control systems are directly labeled as a hierarchical, most systems have at least a two-level hierarchy, with a low-level, real-time subsystem handling real-time motion control, and a slower high-level subsystem responsible for planning. This separation into real-time and non-real-time components has been employed by IBM [KOR86] which has one or more real-time systems, connected to the programming system by a real-time bridge. The University of Pennsylvania has a robot coordinator issue force and motion commands to a real-time robot force and motion server. Brown University uses a series of high-speed networks connecting a real-time servo systems and general computing resources. This basic hierarchical decomposition in each of these applications can be generalized into two virtual control loops exchanging commands and status. IBM extends the exchange of commands and status to include varying frequencies; such as every cycle, every nth invocation or asynchronously. Implicitly, these applications have additional levels within the hierarchy, but these levels do not directly correspond to a virtual control loop implementation because of the concept of one return status implicit in a serial execution of the processes. For example, a Cartesian trajectory plan in a sequential robot language such as AML must be decomposed into a series of kinematic poses for the robot [TAY82]. This process is implicitly hierarchical. These implicit levels could be partitioned into separate concurrent processes that

communicate through established interfaces rather than serial routines that communicate via subroutine calls. Further, with concurrent operation, the concept of sampling the return status, not just receiving a final status, embodies the feedback nature of a virtual control loop.

4.2 Design Architecture at the National Institute of Standards and Technology

The concurrent hierarchical model of a robot control system containing virtual control loops has been implemented with a purely executor style of task decomposition for a robot control system at the National Institute of Standards and Technology [BAR82]. The flow of control was based on state-table transitions. Where planning was appropriate, static plan definitions were used. The system offered several benefits. First, the system was sensory-interactive and adapted to perturbations in the environment in real-time even though the world model was limited to basic feature recognition. Second, hierarchical decomposition created well-defined interfaces that allowed substitution of different implementations of a level without must effect on the higher or lower levels that lead to proposed interface standards [FIT85]. Finally, the system was able to execute in real-time and supply robot updates within a fixed milliseconds interval.

Work has begun on the design and implementation of a concurrent hierarchical control model that includes planning, extensive world modeling including maps, object definitions and feature recognition. The concurrent hierarchical control systems under development will be adapted for both robot and autonomous vehicles. To maintain cost, flexibility and portability, multiprocessors communicating across a shared backplane has been chosen for the parallel architecture. Many commercial products are now available that can handle the multicomputer synchronization and communication problems. The system design is committed to use as many commercial system and support products as possible. Major procurement decisions were based not only on availability and cost, but also on the ability of the product to make as much of the concurrency as transparent to the user as possible.

The system under design is a hybrid design that at run-time clouds the distinction between the host and target system. The real-time part of the system will be termed the target system and is composed of a series of Motorola 68020 processor boards connected via a VME bus. At the heart of the target system is a small, fast, multitasking executive to provide real-time capabilities, but little functionality. This real-time executive does not directly support a file-system. With such a real-time executive, task-switching is in the 100 microsecond range. A system clock synchronizes software execution on all processor boards. This system-wide synchronization pulse forces levels to execute in lock step and allows single-stepping across levels at the fundamental executor cycle rate.

Layered on top of this real-time kernel, is a high-speed multicomputer communication scheme that uses the backplane bus as the physical link. Shared memory globally common to all processors called common memory is the medium in which high speed multicomputer communication takes place. The next layer supports lower-speed communication across backplanes. This transport layer is integrated into the larger network system via either a high-speed, interbucket bus connector, or a local area network. The local area network

protocol is a 4.2BSD Unix[1] socket interface and Transmission Control Protocol/Internet Protocol (TCP/IP). This layer provides services that are non-real-time. The fast networking of the host and target system allows a series of hosts to be used as development systems, and as providers of non-real-time services, such a user-interface or graphic diagnostic display, in the final system configuration.

Figure 10 illustrates the relationship between the various components of the overall system. The target system is partially broken down to illustrate how a level interacts with the system components. Only one level of the hierarchy is displayed and this is but one of several possible scenarios for partitioning the processes among processors. In this case, a partition of the planner and executor across processors uses communication of the plans through shared memory. The world model contains servers to interface to either the planner and executor, and to the sensor processing modules.

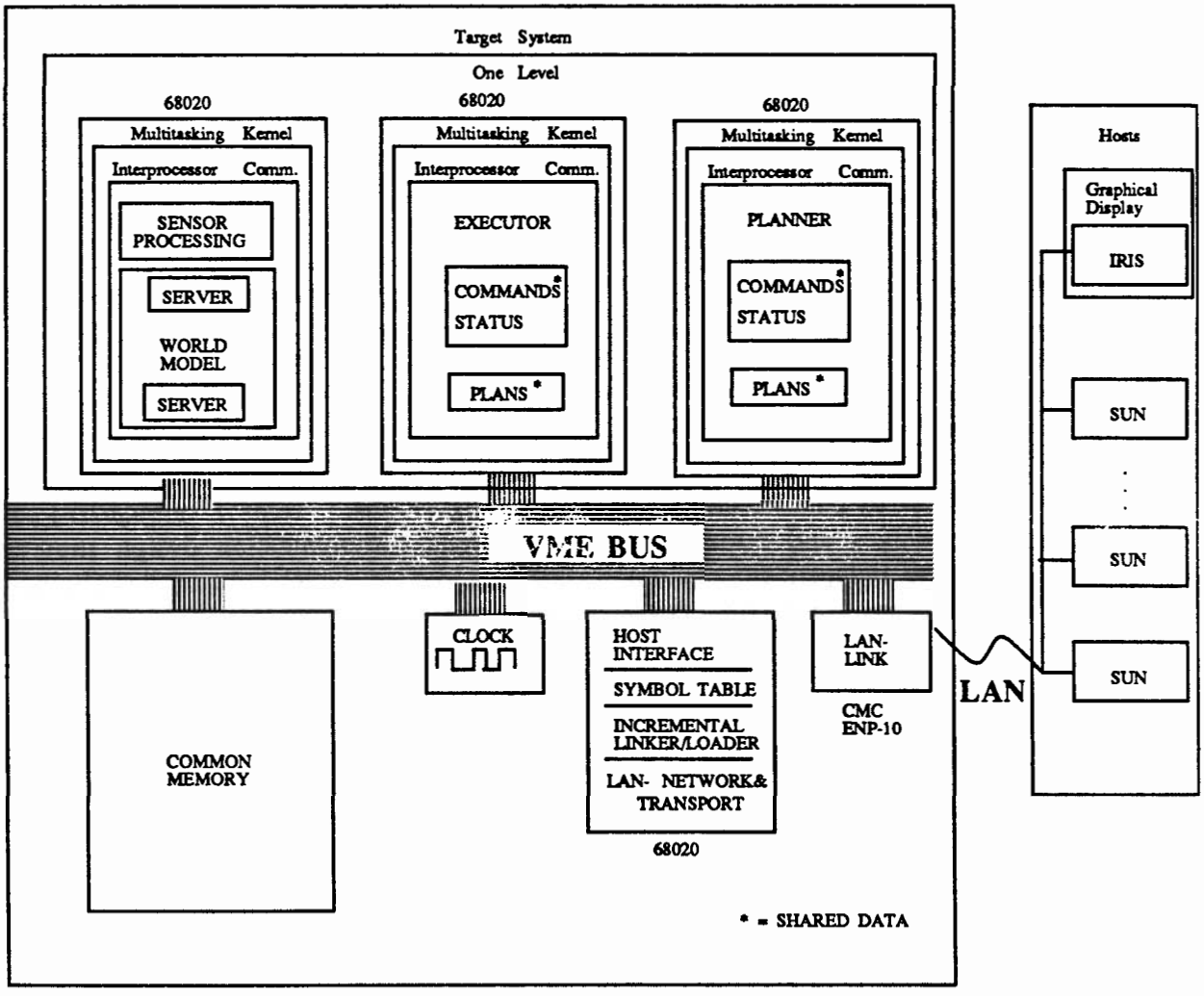

Figure 10. Overview of the System Environment

[1]UNIX is a registered trademark of AT&T Bell Labs

5.0 Conclusion

Just as one cannot use a sequential machine to implement a real-time robot control system, one should not use a sequential methodology to design a real-time robot control system. This paper has addressed some concurrent design issues associated with real-time control. Initially, task decomposition is used in order to generate the system structure hierarchy. The hierarchical model is extended for a real-time control system through the use of concurrent levels within the hierarchy. Each level within the hierarchy emulates a feedback control loop by sampling as inputs the command from a neighboring upper level, comparing the input to the sensory sampled environment, computing a goal-directed action, and outputting this action to the neighboring lower level, all within a fixed response time. This virtual control loop performs periodic sampling of commands and status and insures that global control flow is goal directed. Mapping this concurrent hierarchical model of the control system onto a parallel multicomputing system has proved a reasonable method of implementation. It offers the power of a multicomputer system with the flexibility to assign processes to processors as performance dictates. The concurrent hierarchical software model offers a convenient software structuring tool that allows varying degrees of actual system parallelism.

The impact of concurrency requires new heuristics and rules for the design. A concurrent hierarchical design must first acknowledge that for real-time control, reliable and deterministic performance is the most important design constraint. Mapping concurrency onto a parallel computing environment imposes more demands on the designer. The designer is required to specify how long a module will take and what it will do with unanticipated problems. Finally, multiple computers need interprocessor communication and synchronization to achieve the parallelism. To be effective, the multiple computer communication must be efficient enough to exploit the benefits of parallelism, while minimizing the impact of parallelism on the software algorithms. Table 1 summarizes some of the design advantages and disadvantages of various communication design strategies. Double lines separate independent communication features, of which one of the alternative designs must be chosen. Single lines between the double lines demarcate the selections for each feature.

Issue	Feature	ADVANTAGES	DISADVANTAGES
Coupling	Tightly Coupled Shared Memory	high-speed, simple	Rarely portable, interconnect limits, contention problems
	Loosely Coupled/ Message Passing	Flexible, generic better security, cheaper for larger number of processors	Slower, limited data bandwidth
Synchronization	Synchronous Comm.	Simple, guaranteed comm.	Slow, requires time-out
	Asynchronous Comm.	Flexible, fast	Overhead - queuing
Duration	Temporary Channel	multiplex resources	repeated overhead
	Permanent Channel	low overhead	1:1 commun best,
Connectivity	Dynamic connectivity	flexible, reconfigurable	slower, larger overhead, special hardware
	Static connectivity	simple, fast	centralized server necessary
	Centralized connectivity	better accountability	extra decode overhead each message
	Decentralized connectivity	unbundled, demon model	slower, resource intensive

Table 1. Communication Feature Summary

In summary, augmenting hierarchical structuring of such a control system to contain concurrent control loops offers an easy and systematic approach to generating a parallel model. From experience, an implementation of a hierarchical control system based on these concurrent concepts is both robust and effective because of the structure imposed on the software. The use of levels offers the benefit of information hiding, so that software design and development can concentrate on local problems instead of attempting to solve problems globally. Further, the addition of a system-wide synchronization pulse where the levels execute in lock step adds the dimension of comprehensibility to the system. In general, parallel systems are difficult to understand. With a system heartbeat, execution can be characterized as a state machine where transitions are predictable and repeatable. This does not imply that all software is rigidly defined with a state transition mechanism. Rather, software abstraction is adjustable with selective degrees of resolution; much like changing the magnification of a microscope. At the highest level of abstraction, the state transition are defined as the commands and status exchanged. This allow easy pinpointing of problems within the hierarchy. Tracking execution with a finer resolution of abstraction relies on the basic state transitions employed by the computer. From a software development standpoint, testing and analysis is much easier with a system that allows a selectable resolution of software abstraction. Thus, structuring robot control with a hierarchical model can reduce the complexity of any single processing node and distribute the processing and exploit the parallelism that exists as more and more layers of the control are added.

References

[ALB81] ALBUS, J.S., BARBERA, A.J., NAGEL, R.N. "Theory and Practice of Hierarchical Control," *Twenty-third IEEE Computer Society International Conference*, 1981. pp.18-39.

[ALB87] ALBUS JS, McCAIN H.G., AND LUMIA R. *NASA/NBS Standard Reference Model Telerobot Control System Architecture (NASREM)*, NBS Technical Note 1235, National Bureau of Standards, Gaithersburg, Md., June 1987.

[BAR82] BARBERA, A.J., FITZGERALD, M.L., AND ALBUS, J.S. "Concepts for a Real-Time Sensory-Interactive Control System Architecture," *Proceedings of the 14th Southeastern Symposium on System Theory.* April 1982.

[BRI79] BRITTON, K.H. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," Proceedings of the IEEE Specifications of Reliable Software Conference, 1979.

[DEM79] DEMARCO, T. *Structured Analysis and System Specification.* Yourdan Press, New York, 1979.

[DUB88] DUBOIS, M., SCHEURICH, C. AND BRIGGS, F.A. "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, pp. 9-21, February 1988.

[FIT85] FITZGERALD, M.L., BARBERA, A.J., ALBUS, J.S. "Real-Time Control Systems for Robots," *SPI National Plastics Exposition Conference*, 1985.

[FITZ85] FITZGERALD, M.L., BARBERA, A.J. "A Low-Level Control Interface for Robot Manipulators." NBS-Navy NAV/SIM Workshop of Robots Standards, June 6-7, 1985.

[FRI86] FRIEDLANDER, C.B., AND WEDDE, H.F. "Distributed Processing Under the Dragon Slayer Operating System."

[GAU87] GAUTHIER, D., FREEDMAN, P., CARAYANNIS, G., AND MALOWANY, A.S. "Interprocess Communication for Distributed Robotics," IEEE Journal of Robotics and Automation, Vol. RA03, No. 6, Dec. 1987. pp. 493-504.

[GAG86] GAGLIANELLO, R.D., AND KATSEFF, H.P. "A Distributed Computing Environment for Robotics." In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal. April 1986). IEEE, New York, 1986, pp. 1890-1895.

[GEH86] GEHANI, N.H., ROOME, W.D. "Concurrent C," Software - Practice and Experience, Vol. 16(9), 821-844 (September 1986).

[KAZ87] KAZANZIDES P., WASTI H., AND WOLOVICH W.A. "A Multiprocessor System for Real-Time Robotic Control: Design and Applications," In *Proceedings of the IEEE International Conference on Robotics and Automation* (Raleigh, N.C. March 1987) . IEEE,

New York, 1987, pp. 1903-1908

[KOR86] KOREIN, J.U., MAIER, G.E., TAYLOR, R.H., AND DURFEE, L.F. "A Configurable System for Automation Programming and Control." In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal. April 1986). IEEE, New York, 1986, pp. 1871-1877.

[KRU87] KRUSKAL, C.P., SMITH C.H. "On the Notion of Granularity", National Bureau of Standards Report, July 1987.

[LYO87] LYON, G.E. "On Parallel Processing Benchmarks", National Bureau of Standards Report , NBSIR 87-3580, June 1987. pp.1-23.

[LYO86] LYON, G.E. "Programming The Parallel Processor", Second Symposium on the Role of Language in Problem Solving, sponsored by the Applied Physics Laboratory of Johns Hopkins University, April 2-4, 1986.

[NAR86] NARASIMHAN, S., SIEGEL, D., HOLLERBACH, J.M., BIGGERS, K., AND GERPHEIDE, G. "Implementation of control methodologies on the computational architecture of the Utah/MIT hand." In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal. April 1986). IEEE, New York, 1986, pp. 1884-1889.

[NIL80] NILSSON, N. *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, CA, 1980.

[PAU86] PAUL, R.P, AND ZHANG, H. "Design of a Robot Force/Motion Server." In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal. April 1986). IEEE, New York, 1986, pp. 1878-1883.

[POS80] POSTEL, J. "Internetwork Protocol Approached," IEEE Transactions on Communications, Vol. COM-28, Number 4, April 1980, pp. 604-611.

[SCH85] SCHWAN, K., BIHARI,T., WEIDE, B., AND TAULBEE, G. "GEM: Operating System Primitives for Robots and Real-Time Control Systems," In *Proceedings of the IEEE International Conference on Robotics and Automation* (St. Louis, Mo. March 1985) IEEE, New York, 1985, pp. 807-813.

[SCH87] SCHWAN, K., BIHARI,T., WEIDE, B., AND TAULBEE, G. "High-Performance Operating System Primitives for Robotics and Real-Time Control Systems," *ACM Transactions on Computer Systems*, Vol. 5, No. 3, August 1987, pp. 189-231.

[TAY82] TAYLOR, R.H., SUMMERS, P.D., AND MEYER, J.M. "AML: A Manufacturing Language," *The International Journal of Robotics Research*, Vol. 1, No. 3, Fall 1982, Massachusetts Institute of Technology.

[WIN84] WINSTON, P.H. *Artificial Intelligence*, Addison-Wesley Publishing Company,